Report for shell

Yuming Ding 丁玉铭 2017110267

Contents

1.Abstract

In this experiment, the author implemented a simple command-line interface (CLI) shell using C language.

It can read, decompose, and run commands from the user. Supports the basic functions of shells such as info, exit, pwd, cd, grep, etc. Support for multiple pipes and redirects. Through this experiment, the author skillfully used many system calls under Linux. In practice, I learned about management of processes, inter-process communication and other related knowledge. A deep understanding of the theoretical knowledge within the operating system curriculum.

2.Introdution

In computing, a shell is a user interface for access to an operating system's services. In general, operating system shells use either a command-line interface (CLI) or graphical user interface (GUI), depending on a computer's role and particular operation.[1] It is named a shell because it is the outermost layer around the operating system kernel.[2]

Most operating system shells are not direct interfaces to the underlying kernel, even if a shell communicates with the user via peripheral devices attached to the computer directly. Shells are actually special applications that use the kernel API in just the same way as it is used by other application programs. A shell manages the user–system interaction by prompting users for input, interpreting their input, and then handling an output from the underlying operating system (much like a read–eval–print loop, REPL).[3]

This article focuses on the implementation of a simple CLI shell..CLI shells require the user to be familiar with commands and their calling syntax, and to understand concepts about the shell-specific scripting language (for example bash). They are also more easily operated via refreshable braille display, and provide certain advantages to screen readers.

3.Methodology

3.1 main()

The implementation of this shell is started by the lsh_loop() function in the main function main(). In the loop of the lsh_loop() function, the lsh_read_line() function reads commands from standard input; the lsh_split_line(line) function separates commands by spaces; the lsh_execute(args) function recognizes and executes commands;

3.2

(1)info

Call the **getUsername(username)** function to get the username and save it in *username* and print it.

(2)exit

Return 0. And assign the value to the *status* variable in the **lsh_loop()** function. Terminate the loop and exit the shell program.

(3) pwd

Call the **getcwd(curPath, LSH_RL_BUFSIZE)** function to store the current working path in the variable *curPath*. Then determine the **getcwd ()** function returns the value, if it is empty, the error **perror ("lsh")**; if not empty, print and **return 1**.

(4) cd PATH

First determine if the path parameter of *args[1]* is empty, and if it is empty, it will report a standard error. If it is not empty, call the **chdir(args[1])** function to change the current working directory to the target path pointed to by the *args[1]* parameter. If the **chdir(args[1])** function returns a non-zero value, the change fails and an error is reported. If successful, the **lsh_cd(args)** function **returns 1**.

3.3 lsh_launch(args)

If *args[0]* does not match any of **info, exit, pwd, or cd**, the **lsh_launch(args)** function is called.

In **lsh_launch(args)**, first call the **fork()** function to create a new child process.

If it fails, it will report an error. If it is urrently a child process, call the **dup(STDIN_FILENO)** function to get the file identifier of the standard input, standard output, and then call the **lsh_pipe(args, 0, number_command)** function to process the instruction interval [0,number_command).Finally, **dup2 (in_file_identifier, STDIN_FILENO)** function is called to restore standard input and standard output redirection.

If it is currently a parent process, call the **waitpid(pid, &status, 0)** function to wait for the child process to finish, and finally return the end code of the child process obtained by **WEXITSTATUS()**.

3.4 lsh_pipe(char**args, int left, int right)

In the **lsh_pipe(char**args, int left, int right)** function, first determine whether the pipeline command is included. If the pipeline command is not included, the **lsh_redi(args, left, right)** function is called to process the redirect command. If there is a pipe command, it is judged whether the parameter is complete, and if it is incomplete, an error is reported.

Finally execute the pipeline command. First call the **pipe (file_identifiers)** function to create a pipe, and then call the **vfork ()** function to create a child process. The standard output is redirected to *file_identifiers[1]* in the child process, and then the **lsh_redi(args, left, pipe_position)** function is called to execute a single command before the first pipe. In the parent process, determine whether the instructions of the child process exit normally. If not, an error message is printed. If so, redirect the standard input to *file_identifiers[0]*, then call the **lsh_pipe(args, pipe_position+1, right)** function to recursively execute subsequent instructions.

3.5 lsh_redi(char**args, int left, int right)

In the **lsh_redi(char**args, int left, int right)** function, first determine if there is output redirection. If so, determine the location and number of redirect symbols in *args*. If the number is greater than 1, an error is reported. Call the **vfork()** function to create a child process. If it is currently a child process, if the number of output redirects is 1, then the **freopen(outFile, "w", stdout)** function is called to change the standard output to outFile. Call the **execvp(comm[left], comm+left)** function to run the file. If it is currently a parent process, call **waitpid(pid, &status, 0)** to wait for the child process to finish, and the last **WEXITSTATUS()** gets the child process end code. The return code is not 0, which means that the child process executes an error and prints the error message in red font.

3.6 lsh_grep(char **args)

In the **lsh_grep(char **args)** function, the **textFileRead(args[3])** function is first called to read the text content. Then call the **strindex(text,pattern)** function to count the number of occurrences of the string in the text and print it.

4.Result

A simple shell is implemented in C, which accepts commands entered by the user and performs operations. Support for the built-in command info\exit\pwd\cd PATH\grep. Support for multiple pipes and redirects. After the program runs, the simulated shell displays the current user name, host name, and path in green font, waiting for the user to enter a command. After the program reads the instruction input by the user one by one, the instruction divides the instruction into multiple string commands by space, and then determines the type of the command. If the command is incorrect, the error message is printed in red font.

5. Discussion

5.1 Reason for implementing additional feature.

An additional feature of this shell implementation is the ability to support multiple pipes and redirects. As we all know, in practical applications, single-layer pipelines often fail to meet the actual needs of users. And many shells under the operating system also support multiple pipeline functions. This makes the algorithm for the pipeline in this shell closer to the actual project. Can better exercise programming skills.

At the same time, the shell also simulates the real shell to display the current user name, host name and path in green font. This makes the shell more real.

5.2 The differences between this approach and other possible approaches for some features.

In order to more easily and quickly implement multiple pipelines and redirection functions, the shell's ex, pipeline, and redirection are nested together. This has the advantage of bringing convenience to this programming. And to a certain extent, the difficulty of programming is reduced (it is lazy). A single programmatic also improves the readability of the code. But this approach also has a very big drawback - reducing the maintainability of the code and improving the maintenance cost of the code. If you add some features or modify some features in the next version, this part of the code is likely to face a lot of problems of modification or even rewriting.

So in some other possible ways, it is a more sensible choice to reduce the aggregation and improve the coupling. The ex, pipeline, and redirection functions are implemented separately. Then multiple streams are combined according to the corresponding syntax and rules. This method may seem cumbersome or stupid. But from the perspective of the update iteration of the version is sensible.

5.3 Some differences that would occur if this shell is implemented for a different operating system.

If implemented on the macOS operating system. Because they all support the POSIX protocol. So even though macOS and Linux are very different on the kernel, the difference in using the abstraction layer of system calls is not that great. But if you implement shell functionality for Windows. Because it does not fully support the POSIX protocol. And because Windows is a closed source operating system, it does not disclose its system calls. So programmers can only consult the documentation for Windows API programming. But in fact, the Windows API is just a layer of encapsulation of system calls. To some extent, it also regulates the use of system calls.

6.Reflection and conclusion

Through this implementation of the simple shell. The author has learned a lot, including understanding the basic workings of the shell, improve programming skills and understand how to use system calls， understanding the creation and management of processes， understanding how inter-process communication and resource management can be used.At the same time, it was found that the understanding of the operating system file management system is not very deep and needs to be studied in more depth.

Reference

[1] "The Internet's fifth man", Brain scan, The Economist, London: Economist Group, December 13, 2013, Mr Pouzin created a program called RUNCOM that helped users automate tedious and repeti tive commands. That program, which he described as a “shell” around the computer’s whirring innar ds, gave inspiration—and a name—to an entire class of software tools, called command-line shells, that still lurk below the surface of modern operating systems.

[2]^ a b Raymond, Eric S. (ed.). "shell". The Jargon File.

[3]^ "Operating system shells". AIX 6.1 Information Center. IBM Corp. Retrieved September 16, 201 2.